Google Colab Setup

Please run the code below to mount drive if you are running on colab.

Please ignore if you are running on your local machine.

```
In [ ]: from google.colab import drive
        drive.mount('/content/drive')
```

```
In [ ]: %cd /content/drive/MyDrive/MiniGPT/
```

Language Modeling and Transformers

The project will consist of two broad parts.

1. **Baseline Generative Language Model**: We will train a simple Bigram language model on the text data. We will use this model to generate a mini story.
2. **Implementing Mini GPT**: We will implement a mini version of the GPT model layer by layer and attempt to train it on the text data. You will then load pretrained weights provided and generate a mini story.

Some general instructions

1. Please keep the name of layers consistent with what is requested in the `model.py` file for each layer, this helps us test in each function independently.
2. Please check to see if the bias is to be set to false or true for all linear layers (it is mentioned in the doc string)
3. As a general rule please read the docstring well, it contains information you will need to write the code.
4. All configs are defined in `config.py` for the first part while you are writing the code do not change the values in the config file since we use them to test. Once you have passed all the tests please feel free to vary the parameter as you please.
5. You will need to fill in the `train.py` and run it to train the model. If you are running into memory issues please feel free to change the `batch_size` in the `config.py` file. If you are working on Colab please make sure to use the GPU runtime and feel free to copy over the training code to the notebook.

```
In [ ]: !pip3 install numpy torch tiktoken wandb einops # Install all required packages
```

```
In [1]: %load_ext autoreload
        %autoreload 2
```

```
In [2]: import torch
        import tiktoken
```

```
In [29]: from model import BigramLanguageModel, SingleHeadAttention, MultiHeadAttention, FeedForwardLayer, LayerNorm, TransformerLayer, MiniGPT
         from config import BigramConfig, MiniGPTConfig
         import tests
```

```
In [33]: device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```
In [5]: path_to_bigram_tester = "./pretrained_models/bigram_tester.pt" # Load the bigram model with name bigram_tester.pt
        path_to_gpt_tester = "./pretrained_models/minigpt_tester.pt" # Load the gpt model with name minigpt_tester.pt
```

Bigram Language Model (10 points)

A bigram language model is a type of probabilistic language model that predicts a word given the previous word in the sequence. The model is trained on a text corpus and learns the probability of a word given the previous word.

Implement the Bigram model (5 points)

Please complete the `BigramLanguageModel` class in model.py. We will model a Bigram language model using a simple MLP with one hidden layer. The model will take in the previous word index and output the logits over the vocabulary for the next word.

In [45]:
```
# Test implementation for Bigram Language Model
model = BigramLanguageModel(BigramConfig)
tests.check_bigram(model,path_to_bigram_tester, device)
```

Out[45]:
```
'TEST CASE PASSED!!!'
```

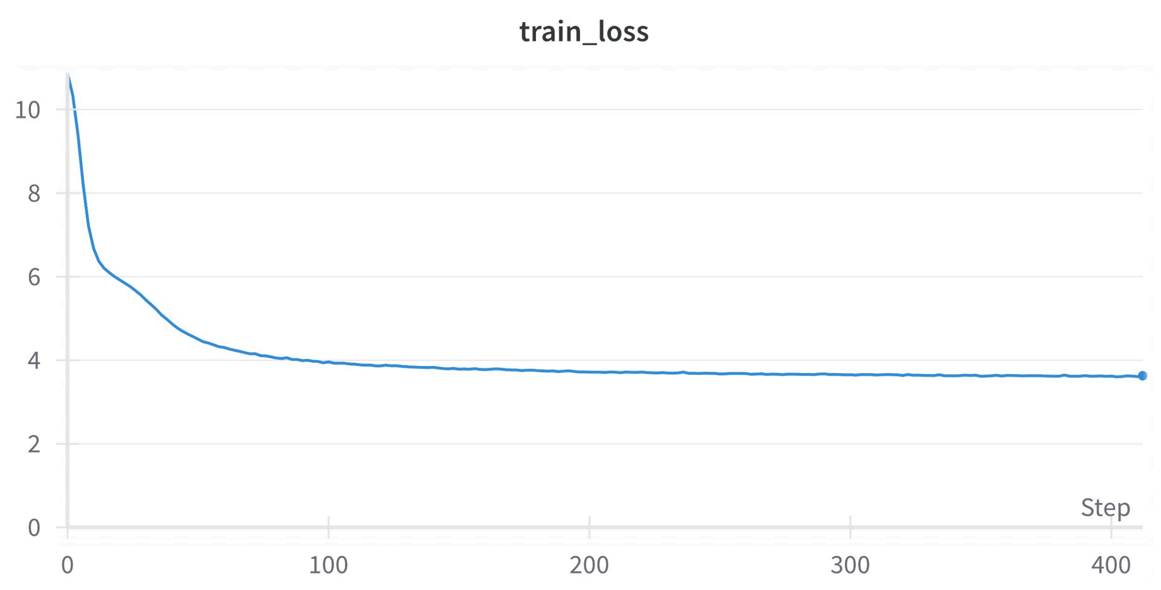
Training the Bigram Language Model (2.5 points)

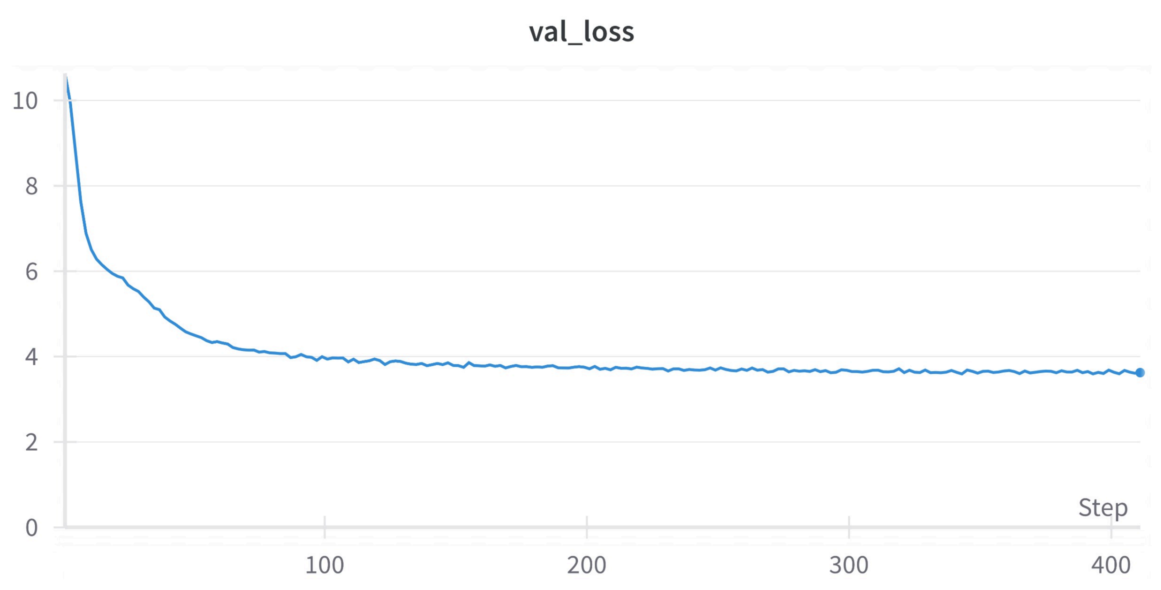
Complete the code in `train.py` to train the Bigram language model on the text data. Please provide plots for both the training and validation in the cell below.

Some notes on the training process:

1. You should be able to train the model slowly on your local machine.
2. Training it on Colab will help with speed.
3. To get full points for this section it is sufficient to show that the loss is decreasing over time. You should see it saturate to a value close to around 5-6 but as long as you see it decreasing then saturating you should be good.
4. Please log the loss curves either on wandb, tensorboard or any other logger of your choice and please attach them below.

Train and Valid Plots





Generation (2.5 points)

Complete the code in the `generate` method of the Bigram class and generate a mini story using the trained Bigram language model. The model will take in the previous word index and output the next word index.

Start with the following seed sentence:

`"once upon a time"`

In [ ]:
```
model_path = "./pretrained_models/model_epoch_1_batch_20600.pth"

# Load the model
model = BigramLanguageModel(BigramConfig)
model.load_state_dict(torch.load(model_path,map_location=torch.device('cpu')))
```

In [31]:
```
tokenizer = tiktoken.get_encoding("gpt2")
```

In [22]:
```
gen_sent = "Once upon a time"
gen_tokens = torch.tensor(tokenizer.encode(gen_sent))
print("Generating text starting with:", gen_tokens.shape)
gen_tokens = gen_tokens.to(device)
model.eval()
print(
    tokenizer.decode(
        model.generate(gen_tokens, max_new_tokens=200).squeeze().tolist()
    )
)
```

```
Generating text starting with: torch.Size([4])
Once upon a time, happy that they could going that it!"
Billy was yummy surprise, they continued to learn knowing she wrote in the drawing. The lion was happy.
Tim, there was glad they all tried to help.
Timmy. They learned that's granted to borrow his back with the tree.
Later that the window and he forgave his friend, butDo you okay, sweet and gave them and saw the yard. He had towers and Tom and made a big smile on't mean. Tweetie was so happy
and said, feeling much fun in key to soar in the story is too hard he never seen things you, there was happy, he forgot about a feather just don't find where it tight. She rode
her mom said goodbye and said, Tim. He then, couldn't know what she heard a different cube before you have my friend. She did not make them for them down and jump and the kitche
n and the cookies. The lady who could not give up the ball
```

Observation and Analysis

Please answer the following questions.

1. What can we say about the generated text in terms of grammar and coherence?
2. What are the limitations of the Bigram language model?
3. If the model is scaled with more parameters do you expect the bigram model to get substantially better? Why or why not?

Answer

1. The generated text has noticeable issues with grammar and coherence. There are many grammatical errors, such as missing punctuation and incomplete sentences. Additionally, the text lacks overall coherence, as the sentences seem disconnected and do not form a coherent narrative.

2. The limitations of the Bigram language model include:

- Lack of context: Bigram models only consider the previous word when predicting the next word, leading to limited contextual understanding.

- Limited vocabulary: Bigram models may struggle with rare or out-of-vocabulary words since they only learn from the observed data.

- Lack of long-range dependencies: Bigram models cannot capture long-range dependencies between words in a sentence, which can result in less accurate predictions.

1. Scaling the Bigram model with more parameters may not substantially improve its performance. Bigram models inherently have limitations due to their simplistic nature, and simply increasing the number of parameters may not address these limitations. Instead, more sophisticated language models, such as recurrent neural networks (RNNs) or transformer-based models like GPT, are better suited for capturing complex language patterns and improving performance on language generation tasks.

Mini GPT (90 points)

We will not implement a decoder style transformer model like we discussed in lecture, which is a scaled down version of the GPT model.

All the model components follow directly from the original Attention is All You Need paper. The only difference is we will use prenormalization and learnt positional embeddings instead of fixed ones. But you will not need to worry about these details!

We will now implement each layer step by step checking if it is implemented correctly in the process. We will finally put together all our layers to get a fully fledged GPT model.

Later layers might depend on previous layers so please make sure to check the previous layers before moving on to the next one.

Single Head Causal Attention (20 points)

We will first implement the single head causal attention layer. This layer is the same as the scaled dot product attention layer but with a causal mask to prevent the model from looking into the future.

Recall that Each head has a Key, Query and Value Matrix and the scaled dot product attention is calculated as :

$$s\text{Attention}(Q, K, V) = \text{softmax}\left(\frac{QK^T}{\sqrt{d_k}}\right)V \tag{1}$$

where d_k is the dimension of the key matrix.

Figure below from the original paper shows how the layer is to be implemented.

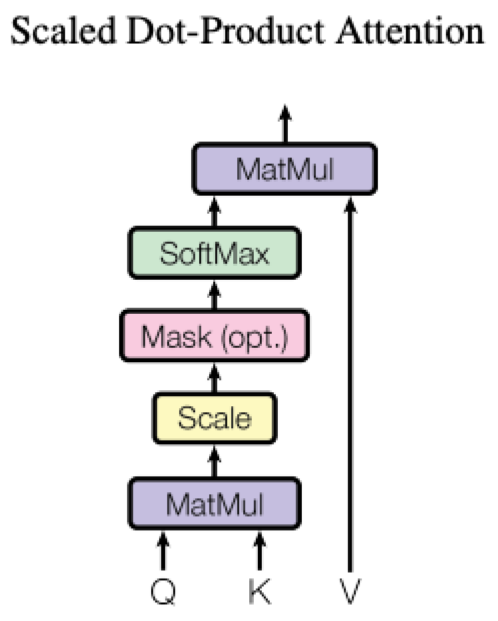


Image credits: Attention is All You Need Paper

Please complete the `SingleHeadAttention` class in `model.py`

In [32]:
```
model = SingleHeadAttention(MiniGPTConfig.embed_dim, MiniGPTConfig.embed_dim//4, MiniGPTConfig.embed_dim//4) # configs are set as such for testing do not modify

tests.check_singleheadattention(model, path_to_gpt_tester, device)
```

Out[32]:
```
'TEST CASE PASSED!!!'
```

Multi Head Attention (10 points)

Now that we have a single head working, we will now scale this across multiple heads, remember that with multihead attention we compute perform head number of parallel attention operations. We then concatenate the outputs of these parallel attention operations and project them back to the desired dimension using an output linear layer.

Figure below from the original paper shows how the layer is to be implemented.

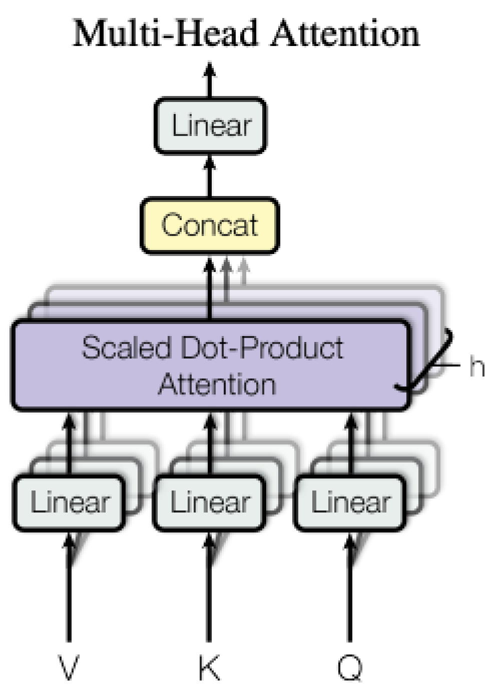


Image credits: Attention is All You Need Paper

Please complete the `MultiHeadAttention` class in `model.py` using the `SingleHeadAttention` class implemented earlier.

In [33]:
```
model = MultiHeadAttention(MiniGPTConfig.embed_dim, MiniGPTConfig.num_heads)

tests.check_multiheadattention(model, path_to_gpt_tester, device)
```

Out[33]:
```
'TEST CASE PASSED!!!'
```

Feed Forward Layer (5 points)

As discussed in lecture, the attention layer is completely linear, in order to add some non-linearity we add a feed forward layer. The feed forward layer is a simple two layer MLP with a GeLU activation in between.

Please complete the `FeedForwardLayer` class in `model.py`

In [36]:
```
model = FeedForwardLayer(MiniGPTConfig.embed_dim)

tests.check_feedforward(model, path_to_gpt_tester, device)
```

Out[36]:
```
'TEST CASE PASSED!!!'
```

LayerNorm (10 points)

We will now implement the layer normalization layer. Layernorm is used across the model to normalize the activations of the previous layer. Recall that the equation for layernorm is given as:

$$\text{LayerNorm}(x) = \frac{x-\mu}{\sqrt{\sigma^2+\epsilon}} \odot \gamma + \beta \tag{2}$$

With the learnable parameters γ and β.

Remember that unlike batchnorm we compute statistics across the feature dimension and not the batch dimension, hence we do not need to keep track of running averages.

Please complete the `LayerNorm` class in `model.py`

In [41]:
```
model = LayerNorm(MiniGPTConfig.embed_dim)
tests.check_layernorm(model, path_to_gpt_tester, device)
```

Out[41]:
```
'TEST CASE PASSED!!!'
```

Transformer Layer (15 points)

We have now implemented all the components of the transformer layer. We will now put it all together to create a transformer layer. The transformer layer consists of a multi head attention layer, a feed forward layer and two layer norm layers.

Please use the following order for each component (Varies slightly from the original attention paper):

1. LayerNorm
2. MultiHeadAttention
3. LayerNorm
4. FeedForwardLayer

Remember that the transformer layer also has residual connections around each sublayer.

The below figure shows the structure of the transformer layer you are required to implement.

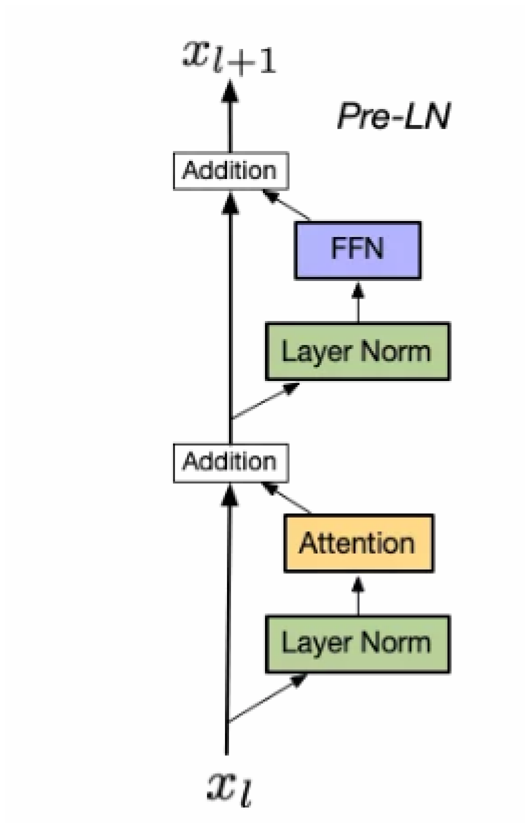


Image Credit : CogView

Implement the `TransformerLayer` class in `model.py`

```
In [43]: model =  TransformerLayer(MiniGPTConfig.embed_dim, MiniGPTConfig.num_heads)
         tests.check_transformer(model, path_to_gpt_tester, device)
```

```
Out[43]: 'TEST CASE PASSED!!!'
```

Putting it all together : MiniGPT (15 points)

We are now ready to put all our layers together to build our own MiniGPT!

The MiniGPT model consists of an embedding layer, a positional encoding layer and a stack of transformer layers. The output of the transformer layer is passed through a linear layer (called head) to get the final output logits. Note that in our implementation we will use weight tying between the embedding layer and the final linear layer. This allows us to save on parameters and also helps in training.

Implement the `MiniGPT` class in `model.py`

```
In [44]: model = MiniGPT(MiniGPTConfig)
         tests.check_miniGPT(model, path_to_gpt_tester, device)
```

```
Out[44]: 'TEST CASE PASSED!!!'
```

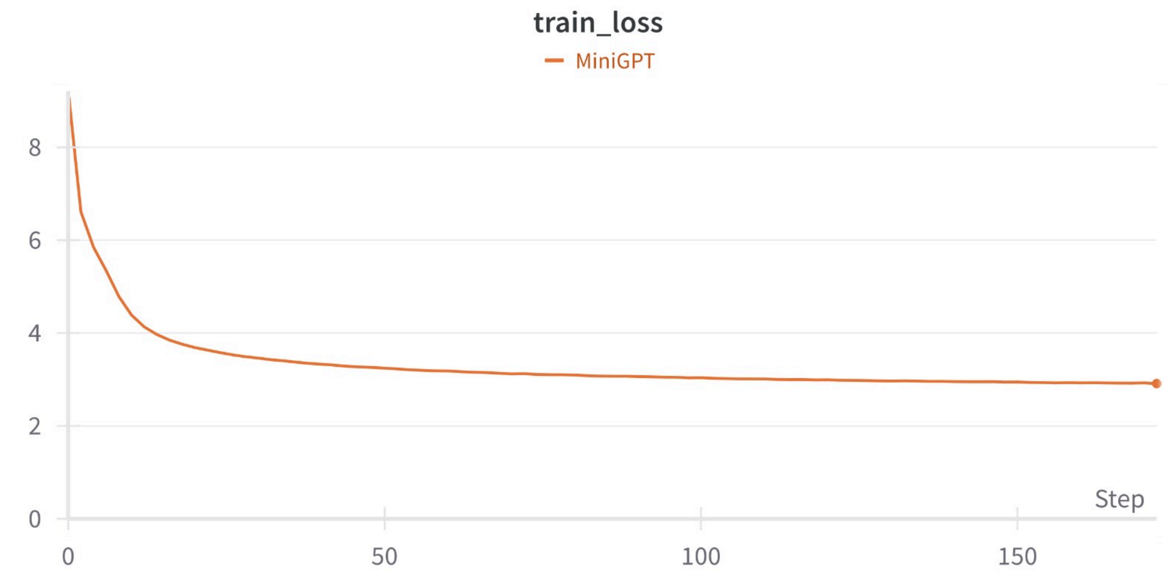
Attempt at training the model (5 points)

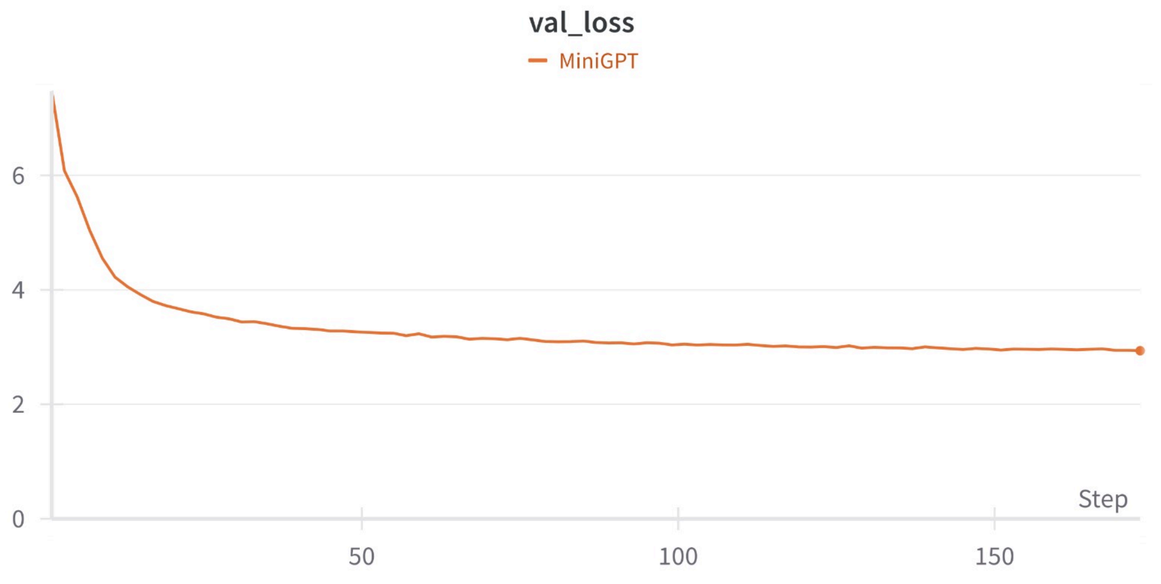
We will now attempt to train the model on the text data. We will use the same text data as before. Please scale down the model parameters in the config file to a smaller value to make training feasible.

Use the same training script we built for the Bigram model to train the MiniGPT model. If you implemented it correctly it should work just out of the box!

NOTE : We will not be able to train the model to completion in this assignment. Unfortunately, without access to a relatively powerful GPU, training a large enough model to see good generation is not feasible. However, you should be able to see the loss decreasing over time. To get full points for this section it is sufficient to show that the loss is decreasing over time. You do not need to run this for more than 5000 iterations or 1 hour of training.

Train and Valid Plots





Generation (5 points)

Perform generation with the model that you trained. Copy over the generation function you used for the Bigram model not the `miniGPT` class and generate a mini story using the same seed sentence.

```
`"once upon a time"`
```

In [ ]:
```
model = MiniGPT(MiniGPTConfig)

model1_path = "./pretrained_models/model_epoch_1_batch_8000.pth"
# Load the model
model.load_state_dict(torch.load(model1_path, map_location=torch.device('cpu')))
```

In [28]:
```
tokenizer = tiktoken.get_encoding("gpt2")

model.to(device)
gen_sent = "Once upon a time"
gen_tokens = torch.tensor(tokenizer.encode(gen_sent))
print("Generating text starting with:", gen_tokens.shape)
gen_tokens = gen_tokens.to(device)
model.eval()
print(
    tokenizer.decode(
        model.generate(gen_tokens, max_new_tokens=200).squeeze().tolist()
    )
)
```

```
Generating text starting with: torch.Size([4])
/Users/sarthakvora/Documents/MiniGPT/model.py:585: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().d
etach().requires_grad_(True), rather than torch.tensor(sourceTensor).

Once upon a time, there was a little boy named Timmy. Timmy loved to play with his toys and sing songs. One day, Max was walking in a house. It drove away and he heard a loud sw
ing. Luna was happy. Then, she played the urge to escape, she tried, but he did. His mother said 'expensive" meant, and the ant. But today he did not find it. Lily remembered th
e kingdom when she went inside her house saw her sitting on the ground. Tom and his dog went to the park. "Do you want to help me." Her mom took a small stick and went to the di
rty room. She knew he would always ask for himself.
One day Mr. Emma was playing. It was careful and still his money. Timmy wanted to eat the special stick." Tom said. "It's a stick!"
Her mom smiled and said, "Yes, I'm always you in yours." Benny was so happy to see the bear and went
```

Please answer the following questions.

1. What can we say about the generated text in terms of grammar and coherence?
2. If the model is scaled with more parameters do you expect the GPT model to get substantially better? Why or why not?

Answer

1. The generated text from the MiniGPT model shows significant improvement in terms of grammar and coherence compared to the Bigram model. The sentences are more grammatically correct, and there is a better overall flow to the narrative. However, there are still some issues with coherence, as some sentences seem disconnected from the overall context or theme.

2. Scaling the MiniGPT model with more parameters is likely to lead to further improvements in its performance. With more parameters, the model can capture more complex patterns in the data and learn better representations of the language. This can lead to more coherent and contextually relevant text generation. Additionally, a larger model may be able to capture longer-range dependencies in the text, leading to more coherent and contextually relevant outputs. However, we do note here that simply increasing the number of parameters is not a guarantee of better performance, and careful optimization is necessary to ensure that the larger model effectively learns from the data.

Scaling up the model (5 points)

To show that scale indeed will help the model learn we have trained a scaled up version of the model you just implemented. We will load the weights of this model and generate a mini story using the same seed sentence. Note that if you have implemented the model correctly just scaling the parameters and adding a few bells and whistles to the training script will results in a model like the one we will load now.

In [47]:
```
from model import MiniGPT
from config import MiniGPTConfig
```

In [48]:
```
path_to_trained_model = "pretrained_models/best_train_loss_checkpoint.pth"
```

In [49]:
```
ckpt = torch.load(path_to_trained_model, map_location=device) # remove map location if using GPU
```

In [50]:
```
# Set the configs for scaled model
MiniGPTConfig.context_length = 512
MiniGPTConfig.embed_dim = 256
MiniGPTConfig.num_heads = 16
MiniGPTConfig.num_layers = 8
```

In [51]:
```
# Load model from checkpoint
model = MiniGPT(MiniGPTConfig)
model.load_state_dict(ckpt["model_state_dict"])
```

Out[51]:
```
<All keys matched successfully>
```

In [52]:
```
tokenizer = tiktoken.get_encoding("gpt2")
```

In [62]:
```
model.to(device)
gen_sent = "Once upon a time"
gen_tokens = torch.tensor(tokenizer.encode(gen_sent))
print("Generating text starting with:", gen_tokens.shape)
gen_tokens = gen_tokens.to(device)
model.eval()
print(
    tokenizer.decode(
        model.generate(gen_tokens, max_new_tokens=200).squeeze().tolist()
    )
)
```

```
Generating text starting with: torch.Size([4])
/Users/sarthakvora/Documents/MiniGPT/model.py:628: UserWarning: To copy construct from a tensor, it is recommended to use sourceTensor.clone().detach() or sourceTensor.clone().d
etach().requires_grad_(True), rather than torch.tensor(sourceTensor).
  #inputs = torch.tensor(generated, dtype=torch.long).unsqueeze(0)
Once upon a time, there was a big, brown dog named Max. Max liked to jog every day. He would jog up and down the hills, up and down the street. Max's owner, a kind lady, would w
ith a gentle smile, a smile.
One day, Max saw a little cat. The cat looked lost and sad. Max wanted to help the cat. He jogged with the cat on his back. The cat was so happy and thanked Max. They became fri
ends and jogged together every day.
One day, the big, brown dog and the cat found a big tree. Max and the cat stopped jogging and looked at the tree. They knew that they had to be careful so they wouldn't fall aga
in. Max and the cat continued jogging and jogged together, feeling proud. They had a lot of fun and learned a lot of new things.Once upon a time there was a little fox. The fox
was very sleepy, so he lay down on a log
```

Bonus (5 points)

The following are some open ended questions that you can attempt if you have time. Feel free to propose your own as well if you have an interesting idea.

1. The model we have implemented is a decoder only model. Can you implement the encoder part as well? This should not be too hard to do since most of the layers are already implemented.
2. What are some improvements we can add to the training script to make training more efficient and faster? Can you concretely show that the improvements you made help in training the model better?
3. Can you implement a beam search decoder to generate the text instead of greedy decoding? Does this help in generating better text?
4. Can you further optimize the model architecture? For example, can you implement Multi Query Attention or Grouped Query Attention to improve the model performance?

Model.py

In [ ]:
```
## Building and training a bigram language model
from functools import partial
import math

import torch
import torch.nn as nn
from einops import einsum, reduce, rearrange


class BigramLanguageModel(nn.Module):
    """
    Class definition for a simple bigram language model.
    """

    def __init__(self, config):
        """
        Initialize the bigram language model.

        The model should have the following layers:
        1. An embedding layer that maps tokens to embeddings. (self.embeddings)
        2. A linear layer that maps embeddings to logits. (self.linear) **set bias to True**
        3. A dropout layer. (self.dropout)

        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
        """

        super().__init__()
        # ========= TODO : START ========= #

        self.embeddings = nn.Embedding(config.vocab_size, config.embed_dim)
        self.linear = nn.Linear(config.embed_dim, config.vocab_size, bias=True)
        self.dropout = nn.Dropout(config.dropout)

        # ========= TODO : END ========= #

        self.apply(self._init_weights)

    def forward(self, x):
        """
        Forward pass of the bigram language model.

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, 1) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, vocab_size) containing the logits.
        """

        # ========= TODO : START ========= #

        x = self.embeddings(x)
        x = self.linear(x)
        out = self.dropout(x)
        return out

        # ========= TODO : END ========= #
    def _init_weights(self, module):
        """
        Weight initialization for better convergence.

        NOTE : You do not need to modify this function.
        """

        if isinstance(module, nn.Linear):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def generate(self, context, max_new_tokens=100):
        """
        Use the model to generate new tokens given a context.
        We will perform multinomial sampling which is very similar to greedy sampling
        but instead of taking the token with the highest probability, we sample the next token from a multinomial distribution.


        Args:
        context : List[int]
            A list of integers (tokens) representing the context.
        max_new_tokens : int
            The maximum number of new tokens to generate.

        Output:
        List[int]
            A list of integers (tokens) representing the generated tokens.
        """

        ### ========= TODO : START ========= ###

        generated = context.tolist()

        with torch.no_grad():
            for _ in range(max_new_tokens):
                inputs = torch.tensor([generated[-1]], dtype=torch.long).unsqueeze(0)
                logits = self.forward(inputs).squeeze(0)
                probabilities = torch.softmax(logits, dim=-1)
                next_token = torch.multinomial(probabilities, num_samples=1).item()
                generated.append(next_token)

        return torch.tensor(generated)


        ### ========= TODO : END ========= ###


class SingleHeadAttention(nn.Module):
    """
    Class definition for Single Head Causal Self Attention Layer.

    As in Attention is All You Need (https://arxiv.org/pdf/1706.03762)

    """

    def __init__(
        self,
        input_dim,
        output_key_query_dim=None,
        output_value_dim=None,
        dropout=0.1,
        max_len=512,
    ):
        """
        Initialize the Single Head Attention Layer.

        The model should have the following layers:
        1. A linear layer for key. (self.key) **set bias to False**
        2. A linear layer for query. (self.query) **set bias to False**
        3. A linear layer for value. (self.value) # **set bias to False**
        4. A dropout layer. (self.dropout)
        5. A causal mask. (self.causal_mask) This should be registered as a buffer.
        NOTE : Please make sure that the causal mask is upper triangular and not lower triangular (this helps in setting up the test cases, )

         NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
        """
        super().__init__()

        self.input_dim = input_dim
        if output_key_query_dim:
            self.output_key_query_dim = output_key_query_dim
        else:
            self.output_key_query_dim = input_dim

        if output_value_dim:
            self.output_value_dim = output_value_dim
        else:
            self.output_value_dim = input_dim

        causal_mask = None  # You have to implement this, currently just a placeholder

        # ========= TODO : START ========= #
        self.key = nn.Linear(input_dim, self.output_key_query_dim, bias=False)
        self.query = nn.Linear(input_dim, self.output_key_query_dim, bias=False)
        self.value = nn.Linear(input_dim, self.output_value_dim, bias=False)

        self.dropout = nn.Dropout(dropout)

        causal_mask = torch.triu(torch.ones(max_len, max_len), diagonal=1).bool()

        # ========= TODO : END ========= #

        self.register_buffer(
            "causal_mask", causal_mask
        )  # Registering as buffer to avoid backpropagation

    def forward(self, x):
        """
        Forward pass of the Single Head Attention Layer.

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, output_value_dim) containing the output tokens.
        """

        # ========= TODO : START ========= #
        # Linear projections
        Q = self.query(x)  # (batch_size, num_tokens, output_key_query_dim)
        K = self.key(x)    # (batch_size, num_tokens, output_key_query_dim)
        V = self.value(x)  # (batch_size, num_tokens, output_value_dim)
        # Scaled dot-product attention
        B, num_tokens, d_k = K.size()
        scores = torch.matmul(Q, K.transpose(-2, -1)) / torch.sqrt(torch.tensor(d_k, dtype=torch.float32))
        scores = scores.masked_fill(self.causal_mask[:num_tokens, :num_tokens] == 1, float('-inf'))
        attn_weights = torch.softmax(scores, dim=-1)
        attn_weights = self.dropout(attn_weights)

        output = torch.matmul(attn_weights, V)

        return output

        # ========= TODO : END ========= #


class MultiHeadAttention(nn.Module):
    """
    Class definition for Multi Head Attention Layer.

    As in Attention is All You Need (https://arxiv.org/pdf/1706.03762)
    """

    def __init__(self, input_dim, num_heads, dropout=0.1) -> None:
        """
        Initialize the Multi Head Attention Layer.

        The model should have the following layers:
        1. Multiple SingleHeadAttention layers. (self.head_{i}) Use setattr to dynamically set the layers.
        2. A linear layer for output. (self.out) **set bias to True**
        3. A dropout layer. (self.dropout) Apply dropout to the output of the out layer.

        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
        """
        super().__init__()

        self.input_dim = input_dim
        self.num_heads = num_heads

        # ========= TODO : START ========= #

        # self.head_{i} = ... # Use setattr to implement this dynamically, this is used as a placeholder
        # self.out = ...
        # self.dropout = ...

        assert input_dim % num_heads == 0, "input_dim must be divisible by num_heads"

        self.head_dim = input_dim // num_heads

        for i in range(num_heads):
            setattr(self, f"head_{i}", SingleHeadAttention(input_dim, self.head_dim, self.head_dim, dropout))

        self.out = nn.Linear(input_dim, input_dim, bias=True)
        self.dropout = nn.Dropout(dropout)

        # ========= TODO : END ========= #

    def forward(self, x):
        """
        Forward pass of the Multi Head Attention Layer.

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.
        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the output tokens.
        """

        # ========= TODO : START ========= #

        # Apply each head's attention
        head_outputs = []
        for i in range(self.num_heads):
            head = getattr(self, f"head_{i}")
            head_output = head(x)
            head_outputs.append(head_output)

        # Concatenate the outputs from all heads
        concatenated = torch.cat(head_outputs, dim=-1)

        # Apply the output linear layer and dropout
        output = self.out(concatenated)
        output = self.dropout(output)

        return output

        # ========= TODO : END ========= #


class FeedForwardLayer(nn.Module):
    """
    Class definition for Feed Forward Layer.
    """

    def __init__(self, input_dim, feedforward_dim=None, dropout=0.1):
        """
        Initialize the Feed Forward Layer.

        The model should have the following layers:
        1. A linear layer for the feedforward network. (self.fc1) **set bias to True**
        2. A GELU activation function. (self.activation)
        3. A linear layer for the feedforward network. (self.fc2) ** set bias to True**
        4. A dropout layer. (self.dropout)

        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
        """
        super().__init__()

        if feedforward_dim is None:
            feedforward_dim = input_dim * 4

        # ========= TODO : START ========= #

        # Define the layers
        self.fc1 = nn.Linear(input_dim, feedforward_dim, bias=True)
        self.activation = nn.GELU()
        self.fc2 = nn.Linear(feedforward_dim, input_dim, bias=True)
        self.dropout = nn.Dropout(dropout)

        # ========= TODO : END ========= #

    def forward(self, x):
        """
        Forward pass of the Feed Forward Layer.

        Args:
        x : torch.Tensor
```

```
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the output tokens.
        """

        ### ========= TODO : START ========= ###

        x = self.fc1(x)
        x = self.activation(x)
        x = self.fc2(x)
        x = self.dropout(x)

        return x

        ### ========= TODO : END ========= ###


class LayerNorm(nn.Module):
    """
    LayerNorm module as in the paper https://arxiv.org/abs/1607.06450

    Note : Variance computation is done with biased variance.
    """

    def __init__(self, normalized_shape, eps=1e-05, elementwise_affine=True) -> None:
        super().__init__()

        self.normalized_shape = (normalized_shape,)
        self.eps = eps
        self.elementwise_affine = elementwise_affine

        if elementwise_affine:
            self.gamma = nn.Parameter(torch.ones(tuple(self.normalized_shape)))
            self.beta = nn.Parameter(torch.zeros(tuple(self.normalized_shape)))

    def forward(self, input):
        """
        Forward pass of the LayerNorm Layer.

        Args:
        input : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the output tokens.
        """

        # ========= TODO : START ========= #

        mean = torch.mean(input, dim=-1, keepdim=True)
        variance = torch.var(input, dim=-1, keepdim=True, unbiased=False)
        normalized_input = (input - mean) / torch.sqrt(variance + self.eps)

        if self.elementwise_affine:
            normalized_input = normalized_input * self.gamma + self.beta

        return normalized_input

        # ========= TODO : END ========= #
```

```python
class TransformerLayer(nn.Module):
    """
    Class definition for a single transformer layer.
    """

    def __init__(self, input_dim, num_heads, feedforward_dim=None):
        super().__init__()
        """
        Initialize the Transformer Layer.
        We will use prenorm layer where we normalize the input before applying the attention and feedforward layers.

        The model should have the following layers:
        1. A LayerNorm layer. (self.norm1)
        2. A MultiHeadAttention layer. (self.attention)
        3. A LayerNorm layer. (self.norm2)
        4. A FeedForwardLayer layer. (self.feedforward)

        NOTE : PLEASE KEEP OF EACH LAYER AS PROVIDED BELOW TO FACILITATE TESTING.
        """

        # ========= TODO : START ========= #

        # Define the layers
        self.norm1 = LayerNorm(input_dim)
        self.attention = MultiHeadAttention(input_dim, num_heads)
        self.norm2 = LayerNorm(input_dim)
        self.feedforward = FeedForwardLayer(input_dim, feedforward_dim)

        # ========= TODO : END ========= #

    def forward(self, x):
        """
        Forward pass of the Transformer Layer.

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, num_tokens, token_dim) containing the output tokens.
        """

        # ========= TODO : START ========= #

        x_norm1 = self.norm1(x)
        attention_output = self.attention(x_norm1)
        x = x + attention_output

        x_norm2 = self.norm2(x)
        feedforward_output = self.feedforward(x_norm2)
        x = x + feedforward_output

        return x

        # ========= TODO : END ========= #


class MiniGPT(nn.Module):
    """
    Putting it all together: GPT model
    """

    def __init__(self, config) -> None:
        super().__init__()
        """
        Putting it all together: our own GPT model!

        Initialize the MiniGPT model.

        The model should have the following layers:
        1. An embedding layer that maps tokens to embeddings. (self.vocab_embedding)
        2. A positional embedding layer. (self.positional_embedding) We will use learnt positional embeddings.
        3. A dropout layer for embeddings. (self.embed_dropout)
        4. Multiple TransformerLayer layers. (self.transformer_layers)
        5. A LayerNorm layer before the final layer. (self.prehead_norm)
        6. Final language Modelling head layer. (self.head) We will use weight tying (https://paperswithcode.com/method/weight-tying) and set the weights of the head layer to be

        NOTE: You do not need to modify anything here.
        """

        self.vocab_embedding = nn.Embedding(config.vocab_size, config.embed_dim)
        self.positional_embedding = nn.Embedding(
            config.context_length, config.embed_dim
        )
        self.embed_dropout = nn.Dropout(config.embed_dropout)

        self.transformer_layers = nn.ModuleList(
            [
                TransformerLayer(
                    config.embed_dim, config.num_heads, config.feedforward_size
                )
                for _ in range(config.num_layers)
            ]
        )

        # prehead layer norm
        self.prehead_norm = LayerNorm(config.embed_dim)

        self.head = nn.Linear(
            config.embed_dim, config.vocab_size
        )  # Language modelling head

        if config.weight_tie:
            self.head.weight = self.vocab_embedding.weight

        # precreate positional indices for the positional embedding
        pos = torch.arange(0, config.context_length, dtype=torch.long)
        self.register_buffer("pos", pos, persistent=False)

        self.apply(self._init_weights)

    def forward(self, x):
        """
        Forward pass of the MiniGPT model.

        Remember to add the positional embeddings to your input token!!

        Args:
        x : torch.Tensor
            A tensor of shape (batch_size, seq_len) containing the input tokens.

        Output:
        torch.Tensor
            A tensor of shape (batch_size, seq_len, vocab_size) containing the logits.
        """

        ### ========= TODO : START ========= ###
        batch_size, seq_len = x.size()

        token_embeddings = self.vocab_embedding(x)

        pos_embeddings = self.positional_embedding(self.pos[:seq_len])

        embeddings = token_embeddings + pos_embeddings

        embeddings = self.embed_dropout(embeddings)

        for layer in self.transformer_layers:
            embeddings = layer(embeddings)

        embeddings = self.prehead_norm(embeddings)

        logits = self.head(embeddings)

        return logits

        ### ========= TODO : END ========= ###

    def _init_weights(self, module):
        """
        Weight initialization for better convergence.

        NOTE : You do not need to modify this function.
        """

        if isinstance(module, nn.Linear):
            if module._get_name() == "fc2":
                # GPT-2 style FFN init
                torch.nn.init.normal_(
                    module.weight, mean=0.0, std=0.02 / math.sqrt(2 * self.num_layers)
                )
            else:
                torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)
            if module.bias is not None:
                torch.nn.init.zeros_(module.bias)
        elif isinstance(module, nn.Embedding):
            torch.nn.init.normal_(module.weight, mean=0.0, std=0.02)

    def generate(self, context, max_new_tokens=100):
        """
        Use the model to generate new tokens given a context.

        Please copy the generate function from the BigramLanguageModel class you had implemented earlier.
        """

        ### ========= TODO : START ========= ###

        generated = context.tolist()

        with torch.no_grad():
            for _ in range(max_new_tokens):
                inputs = torch.tensor(context, dtype=torch.long).unsqueeze(0)


                logits = self.forward(inputs).squeeze(0)
                probabilities = torch.softmax(logits, dim=-1)
                next_token = torch.multinomial(probabilities, num_samples=1)#.item()

                generated.append(next_token[-1].item())
                if len(context) < 10:
                    context = torch.cat((context, next_token[-1]))
                else:
                    context = torch.cat((context[1:], next_token[-1]))


        return torch.tensor(generated)


        ### ========= TODO : END ========= ###
```

Train.py

In [ ]:
```
from pathlib import Path
import random
import torch
import torch.nn as nn
import torch.optim as optim
from torch.utils.data import DataLoader
from einops import rearrange
import wandb

from model import BigramLanguageModel, MiniGPT
from dataset import TinyStoriesDataset
from config import BigramConfig, MiniGPTConfig
from tqdm import tqdm

MODEL = "minigpt"  # bigram or minigpt

if MODEL == "bigram":
    config = BigramConfig
    model = BigramLanguageModel(config)
elif MODEL == "minigpt":
    config = MiniGPTConfig
    model = MiniGPT(config)
else:
    raise ValueError("Invalid model name")

# Initialize wandb if you want to use it
if config.to_log:
    wandb.init(project="dl2_proj3")

def count_parameters(model):
    return sum(p.numel() for p in model.parameters() if p.requires_grad)

train_dataset = TinyStoriesDataset(
    config.path_to_data,
    mode="train",
    context_length=config.context_length,
)
eval_dataset = TinyStoriesDataset(
    config.path_to_data, mode="test", context_length=config.context_length
)

train_dataloader = DataLoader(
    train_dataset, batch_size=config.batch_size, pin_memory=True
)
eval_dataloader = DataLoader(
    eval_dataset, batch_size=config.batch_size, pin_memory=True
)

device = torch.device("cuda" if torch.cuda.is_available() else "cpu")
```

```python
print("number of trainable parameters: %.2fM" % (count_parameters(model) / 1e6,))

if not Path.exists(config.save_path):
    Path.mkdir(config.save_path, parents=True, exist_ok=True)

# Loss function and optimizer
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=config.learning_rate)

model.to(device)

# Function to evaluate on a random subset of validation set
def evaluate(model, eval_dataloader, criterion, subset_size=10):
    model.eval()
    eval_loss = 0.0
    count = 0

    eval_iter = iter(eval_dataloader)
    sampled_batches = 0
    while sampled_batches < subset_size:
        try:
            inputs, targets = next(eval_iter)
        except StopIteration:
            break  # No more batches to sample from

        inputs, targets = inputs.to(device), targets.to(device)
        with torch.no_grad():
            outputs = model(inputs)
            loss = criterion(outputs.view(-1, config.vocab_size), targets.view(-1))
            eval_loss += loss.item()
            count += 1
            sampled_batches += 1

    return eval_loss / count if count > 0 else float('inf')

# Training loop
model.train()
for epoch in range(1):
    running_loss = 0.0
    for i, (inputs, targets) in enumerate(tqdm(train_dataloader, desc="Training")):
        inputs, targets = inputs.to(device), targets.to(device)

        optimizer.zero_grad()
        outputs = model(inputs)
        loss = criterion(outputs.view(-1, config.vocab_size), targets.view(-1))
        loss.backward()
        optimizer.step()

        running_loss += loss.item()

        if i % config.log_interval == 0 and i > 0:
            avg_loss = running_loss / config.log_interval
            print(f"Epoch [{epoch+1}], Batch [{i}], Loss: {avg_loss:.4f}")

            # Log the loss to wandb
            if config.to_log:
                wandb.log({"train_loss": avg_loss})

            # Evaluate on a subset of the validation set
            eval_loss = evaluate(model, eval_dataloader, criterion)
            print(f"Validation Loss: {eval_loss:.4f}")

            # Log the validation loss to wandb
            if config.to_log:
                wandb.log({"val_loss": eval_loss})

            running_loss = 0.0

            # Save the model
            if i % config.save_iterations == 0:
             torch.save(model.state_dict(), config.save_path / f"model_epoch_{epoch+1}_batch_{i}.pth")

print("Training complete")
```